CS 371 Pong Project

Caroline Waters & AJ Wyatt Visit the GitHub here

Background

The problem at hand is the design of a multiplayer Pong game utilizing client-server architecture. Players (the clients) play the game over a network, facilitated by a single server managing the client connection. The implementation of this architecture is done through Python socket programming.

Design

We chose to design this project using a single server handling two client threads. Each client has its own information about game state and synchronization, which is sent to the server. The server then determines which client has the most up-to-date game state and corrects the state and synchronization accordingly. Only the clients are responsible for playing the game. The server manages both client threads (and client connections) simultaneously.

Implementation (see UML below)

pongServer.py

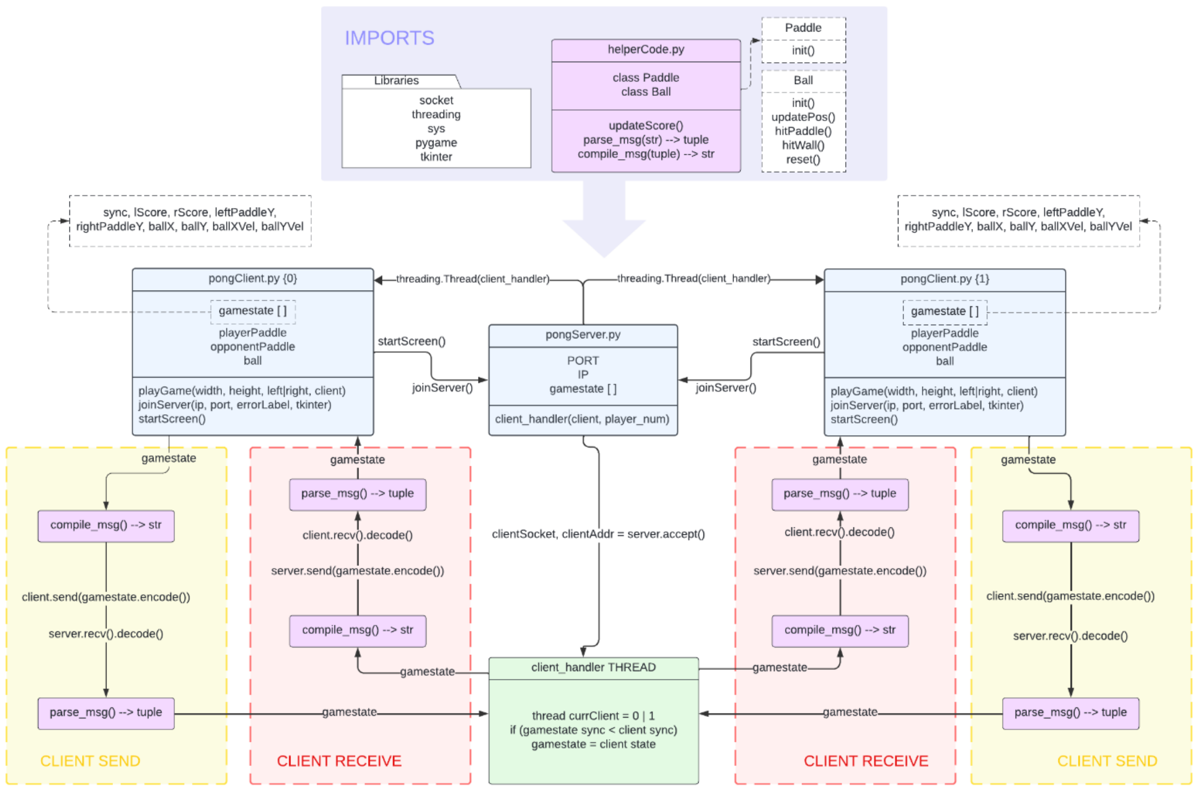
Maintains the most current game states, server IP and port information, as well as threads and thread locks. Listens for clients to join, and then accepts client join requests in order to start two threads. The client_handler function handles the threads, and takes in the client socket as well as the player number (0 or 1) as an input. Client_handler assigns each player as either left or right, and sends this string to the clients. During the game loop, the server receives updates from both clients, and evaluates the sync value to determine which update is most current. The critical thread section in client_handler does this check, and then assigns the server's game state to the most current value. Then, it sends an update to the clients with this game state. At the end of client_handler, the client connections are closed.

pongClient.py

Responsible for game logic and connecting to server. Generates windows visible when code is run. There must be two pongClient.py programs running simultaneously for this architecture to function. When run, calls startScreen(), which generates the window to input server IP and port address to join a game. Calls joinServer(), which takes IP, port, errorLabel, and app (tkinter attributes). joinServer() is responsible for creating a client socket, connecting it to the server, and then receiving information about which side the client is playing as. It then closes the join game window, and calls playGame() which takes the screen dimensions (hard coded as 960 x 720), "left" or "right", and the client socket. playGame() is responsible for the actual gameplay. It initializes the screen, some game constants, and the size and appearance of the objects in the screen to certain defaults. It then performs a while loop which continuously updates the game window, sends the server an update on its current game state, and then updates itself from the most current game state sent by the server. Each iteration of the loop also updates the sync by 1, and calls clock.tick(60) which runs the game at 60 frames per second.

helperCode.py

We added compile_msg() and parse_msg() to facilitate data transfer. Compile_msg() takes a tuple and turns it into a string to be sent, and parse_msg() takes a string and turns it into a tuple in order to access individual values for game state updates.



Challenges

The primary challenges for implementation were our newness to Python and the Python socket library. Small differences between functions like send() and sendall() often had large impacts on functionality, and documentation was sometimes lacking in specifics. Another hurdle was figuring out how to send and receive the game state values, since the code examples we had been given in class just involved small strings.

General debugging was made more difficult due to the nature of the project. Trying to run the Visual Studio debugger on two separate clients simultaneously often yielded misleading results, since the synchronization could change based on who stepped through the statements faster. The threading also made errors more confusing since it was hard to tell if/where an error occurred in a thread. When running two clients and a server program, performance tended to vary based on if each program had its own device versus two clients being run on the same computer. Similarly, some errors would only occur on one device. The synchronization might work on the client that joined first, while the second client would not work at all.

Lessons Learned

We both have gained a significant understanding of the client-server architecture on a lower level. Being able to tinker with the socket library revealed a lot about what goes on “under the hood” of a client-server connection, like the actual sending and receiving of data through buffers, or how connections are established and handled. The task of implementing a synchronization strategy for the two clients showed us more about how real-time connections between multiple clients are maintained, and we were able to apply our understanding of connection-oriented reliability (or lack thereof) to a concrete example. Additionally, the use of threads in a more practical context than previous class examples helped us draw more conclusions about how threading is utilized in real life applications.

Known Bugs

The ball occasionally gets past the top or bottom screen border if it is moving fast enough. This issue persists even when playing locally on a computer without using a network connection, so it is not caused by server lag. An issue that similarly appears on a non-networked connection is that the ball gains massive speed if it hits the top or bottom edge of the paddle. Aside from that, there are frequent network lag spikes that cause balls or paddles to not update appropriately, which could probably be fixed if we had more time to optimize the way data is sent.

Conclusions

The implementation of a Pong game using client-server architecture allowed us to gain insight into real-world client-server applications, as well as extend our networking knowledge from the classroom to a practical project. In the future, it would be an interesting challenge to extend this type of framework to accommodate more than two clients at once to learn more about how modern multiplayer games handle several connections.